

AD-A279 855



NPSCS-94-003

NAVAL POSTGRADUATE SCHOOL
Monterey, California



DTIC
ELECTE
MAY 26 1994
S F D

94-15758



**SEMIAUTOMATIC DEABBREVIATION OF
SOURCE PROGRAMS**

by Neil C. Rowe¹ and Kari Laitinen

March 1994

Approved for public release; distribution is unlimited.

Prepared for:

DARPA
3701 N. Fairfax Drive
Arlington, VA 22203-1714

Naval Postgraduate School
Monterey, CA 93943-5118

94 5 25 050

NAVAL POSTGRADUATE SCHOOL
Monterey, California

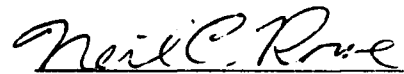
REAR ADMIRAL T. A. MERCER
Superintendent

HARRISON SHULL
Provost

This work was sponsored by DARPA as part of the 13 Project under AO 8939, and by the Naval Postgraduate School under funds provided by the Chief for Naval Operations

Reproduction of all or part of this report is authorized.

This report was prepared by:



Neil C. Rowe
Professor of Computer Science

Reviewed by:

Released by:



Yutaka Kanayama
Associate Chairman for
Technical Research



PAUL J. MARTO
Dean of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE				
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPSCS-94-003			5. MONITORING ORGANIZATION REPORT NUMBER(S) Naval Postgraduate School	
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School		6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION DARPA	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943			7b. ADDRESS (City, State, and ZIP Code) 3701 N. Fairfax Drive Arlington, VA 22203-1714	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Postgraduate School		8b. OFFICE SYMBOL (if applicable) NPS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AO 8939	
8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943			10. SOURCE OF FUNDING NUMBERS	
			PROGRAM ELEMENT NO.	PROJECT NO.
			TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Semiautomatic Deabbreviation of Source Programs				
12. PERSONAL AUTHOR(S) Neil C. Rowe and Kari Laitinen				
13a. TYPE OF REPORT Progress		13b. TIME COVERED FROM Jan 93 TO Dec 93	14. DATE OF REPORT (Year, Month, Day) 1994 3	15. PAGE COUNT 10
16. SUPPLEMENTARY NOTATION				
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) software engineering, abbreviations, tools, comprehensibility, understand- ability, query optimization	
FIELD	GROUP	SUB-GROUP		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Evidence suggests that using variable and procedure names consisting of whole natural-language words helps program comprehensibility. We describe a tool to help users make their programs more comprehensible and thus maintainable by suggesting replacements for the abbreviations in the programs. Its heuristics limit the search for possible "deabbreviations" to just a few good guesses. This is done by examining words in program comments and in a large English dictionary to recognize pieces of English words within multiword abbreviations. Experimental results show the tool is easy to use and results in significantly improved program comprehensibility.				
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Neil C. Rowe			22b. TELEPHONE (Include Area Code) (408) 656-2462	22c. OFFICE SYMBOL CSRp

Semiautomatic Deabbreviation of Source Programs

Neil C. Rowe¹ and Kari Laitinen

Department of Computer Science

Code CS/Rp, U. S. Naval Postgraduate School

Monterey, CA USA 93943

(rowe@cs.nps.navy.mil)

ABSTRACT

Evidence suggests that using variable and procedure names consisting of whole natural-language words helps program comprehensibility. We describe a tool to help users make their programs more comprehensible and thus maintainable by suggesting replacements for the abbreviations in the programs. Its heuristics limit the search for possible "deabbreviations" to just a few good guesses. This is done by examining words in program comments and in a large English dictionary to recognize pieces of English words within multiword abbreviations. Experimental results show the tool is easy to use and results in significantly improved program comprehensibility.

¹ This work was sponsored by the Defense Advanced Research Projects Administration, as part of the I3 Project under AO 8939, and by the Technical Research Centre of Finland (VTT).

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced <input type="checkbox"/>	
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

1. Introduction

Abbreviations of natural-language words are common in computer programs, most frequently for procedure and variable names. Some evidence suggest that abbreviations, as opposed to the full natural-language words from which they are derived, adversely affect comprehensibility of a source program. [Schneiderman, 1980, pages 70-75] and [Weissman, 1974] showed experiments in which significant improvement in ability to answer questions about a program occurred when mnemonic names were used instead of abbreviated names in the program, and [Laitinen, 1992] showed programmers preferred to work with programs having natural-language-word or "natural" names. Natural names can be developed early in the software design cycle by a structured methodology like [Laitinen and Mukari, 1992]. But not all programmers will accept this additional imposition, and more importantly, this methodology cannot be used with existing programs that need maintenance. So we have explored semi-automatic replacement of abbreviations by natural names in existing programs and have developed a prototype tool. This new tool for software engineering we call a "deabbreviator". Its output can either replace the original program or serve as an aid to its understanding, facilitating subsequent modification or reuse of the program.

Some simple tools replace abbreviations according to a fixed list, including DataLift from Peoplesmith, Inc., Magic Typist from Olduvai Corp., and The Complete Writer's Toolkit from Systems Compatibility Corp. [Laitinen, 1992] mentions a more sophisticated tool that also checks for conflicts among replacement names. Unfortunately, abbreviations vary considerably between applications, and the general-purpose replacement lists of these programs are of only limited help for the often specialized task of software description. For example, "lbr" is common abbreviation for "labor" in business, but not in a biological laboratory. [Rosenberg, 1992] lists an average of two different interpretations for every three-word abbreviation, even for the restricted domain of "information technology" and the restriction to common interpretations. An improvement would be application-specific abbreviation pairs plus "deabbreviation" rules that can intelligently guess replacements not in the list. But the challenge is to find a way to limit the seemingly unbounded number of words that could be checked as the source of

an abbreviation.

2. Data structures

Our deabbreviation tool uses the following support files, all hashed:

- a dictionary list of 29,000 common English words;
- a list of reserved words for the programming language and/or operating system whose programs are being analyzed;
- an auxiliary list of valid English words found in previous runs but not in the preceding files;
- common words of the application domain, used as additional comment words (optional);
- standard abbreviations used in computer software, with their deabbreviations;
- replacements accepted for previous programs (optional).

The dictionary is necessary to define the acceptable natural names in a program. To obtain it, we combined wordlists from the Berkeley Unix "spell" utility, the GNU Emacs "ispell" utility, the first author's papers, some saved email, and captions from 100,000 pictures at a Navy Laboratory. We removed abbreviations from these sources manually, and were particularly liberal in removing words under four letters in length. This gave us a reasonably broad vocabulary of about 29,000 English words. This dictionary is then supplemented with reserved words specific to the programming language being used, and any new words confirmed by the user on previous runs. The user may also include a optional file of words specific to the real-world domain that the program addresses; these can duplicate the dictionary, but have special priority in trying to find deabbreviations.

We also manually compiled a "standard abbreviation list" of 168 entries from a study of programs written by a variety of programmers, supplemented by the list of [Rosenberg, 1992]; example entries are "ptr -> pointer" and "term -> terminal". These are the abbreviations we observed repeatedly in a wide range of program examples. The list was kept short to reduce domain-dependent alternatives; and it was not made any shorter because explicit listing saves time with common abbreviations, although many entries could be derived from the abbreviation rules discussed next. The list is supplemented by

deabbreviations confirmed or provided by the user, as we will discuss.

3. Deabbreviation methods

Deabbreviating means replacing an abbreviated name with a more understandable "natural" one consisting of whole English words. To deabbreviate, we "generate-and-test": We select candidates and abbreviate them various ways, trying to obtain a match with a given abbreviation. The abbreviation rules derive mostly from our study of example programs, with some ideas from [Bourne and Ford, 1961]. Three "word abbreviation" methods are used: (1) match to a standard deabbreviation, previous user replacement, or "analogy" (see below), as confirmed by lookup in the corresponding hash table; (2) truncation on its right end of some word in the program's comments, as long as at least two letters remain and at least two letters are eliminated; and (3) elimination of vowels in a comment word, provided at least two letters remain including the first. Then a "full abbreviation" is either one, two, or three "word abbreviations" appended together. These methods together explain about 98 percent of abbreviations we observed in example programs, as abbreviations rarely get more complicated than three-word, although we do have ways to find longer abbreviations as discussed below. For an m -letter word, word-abbreviation method (1) generates $O(1)$ abbreviations for matching, method (2) $O(m)$, and method (3) $O(1)$; so the whole algorithm generates $O(m^3)$ full abbreviations. Cubic behavior was observed to be the limit if processing were to be fast enough.

A key feature of our approach is the restriction of word-abbreviation methods (2) and (3) to words in the comments on the source program, supplemented by optional application-domain words. With at 29000-word dictionary, there would be 25 million million combinations to abbreviate otherwise. We believe that if a program is properly commented, the comments should contain most of the natural names that would be appropriate to use in its variable and procedure names beyond the standard programming words like "pointer". But we do not insist that comment words be near their abbreviations, as comments can have global scope and can reference backward as well as forward. Note that word-abbreviation methods (2) and (3) leave intact the initial letter of a word or phrase, consistent with the

programs we studied, and that eliminates many possibilities.

Since a word can have many abbreviations, the nondeterminism of Prolog is valuable, as it permits backtracking with depth-first search to get further deabbreviations until the user finds one acceptable.

The above ordering of the methods is used heuristically in this depth-first search, so that direct hash lookups are tried before deleting characters, single-word abbreviations before two-word, and two-word before three-word. These heuristics derive from cost analysis of the methods. With two-word and three-word methods, an additional heuristic, based on study of example programs, considers rightmost splits first, so "tempvar" would be first split into "tempv" and "ar", then "temp" and "var".

4. Control structure

The control structure of the deabbreviation tool has similarities to those of spelling correctors [Peterson, 1980], text-error detectors [Kukich, 1992], and copy-editor assistants like [Dale, 1989]. But abbreviated words are farther from their sources than misspelled words are, so a deabbreviator requires more computation per word; and abbreviating loses so much information that it cannot be inverted and we must rely on generate-and-test.

Processing has three passes. On the first pass, the input program is read in, comment words more than 3 characters are stored, and non-comment words are looked up in the dictionary. Reserved words of the programming language, words within quotation marks, words containing numbers, and words already analyzed are ignored. Morphology and case considerations complicate the lookup. To simplify matters, our dictionary and standard abbreviations are kept in lower case, and comment and unknown non-comment words are converted to lower case for matching. Similarly, most of the dictionary is minus the standard English suffixes "s", "ed", and "ing", and unknown words for comparison are stripped of any such endings, using the appropriate morphological rules of English for undoubling of final consonants, adding a final "e", and changing "i" to "y". The unknown word and all stripped words are tried separately, to catch misleading words like "string" and "fuss".

The second pass then individually considers the words that did not match on the first pass. Full abbreviations are generated to try to match the given word. Plurals are also tried to match to the given word or pieces of it. Possible deabbreviations are individually displayed to the user for approval; if the user rejects them all, the user is asked if the word can be added to the dictionary. If not, the user must supply a replacement, which is recursively checked to contain only dictionary words, allowing underscores for punctuation. Replacements can be specified either global for the entire program or local to the procedure definition.

We considered ranking of the alternative deabbreviations for a word during the second pass, and presenting the most likely to the user first. However, we concluded that this was not cost-effective. It would require about 10 seconds per unknown word in our current implementation. To do it properly, we must determine the frequency of every one of the 29,000 words in the dictionary, and frequencies would vary with domain. Abbreviation methods also vary considerably in frequency even with the same programmer.

The third pass changes the names in the program as decided on the second pass. Name collisions can occur if user-approved deabbreviations are identical with words already used in the program, and the user is then given a warning.

5. Analogies

Replacements can also be indirectly inferred from analogies, and this is one of the most important features of the tool. For instance, if the user confirms that they want to replace "buftempfoobar" with "buffer_temporary_foobar", we can infer that a replacement for "buf" is "buffer" and for "temp" is "temporary", even if we do not recognize "buf" and "foobar". Previous user replacements can be found within an analogy too, to simplify matters, and multiple deabbreviations of the same abbreviation must be permitted to be inferred (although at lesser priority than user-sanctioned deabbreviations). We discovered that analogies are very helpful in deabbreviating real programs because user names are often interrelated. Such analogies should be found after a replacement is approved by the user, not during the

generation of deabbreviations when there would be too many possibilities to consider. Caching of analogies that are rejected is important to prevent requerying them.

Replacements can also be inferred from analogies between two previous replacements. For instance, if the user confirmed that "temp" should be replaced by "temporary", and then confirmed "tempo" should be replaced by "temporary_operator", we should infer that one possible replacement for "o" in this program is "operator" even though "tempo" is an English word. "tempo" is a truncation of "temporary", and one-word abbreviations are quite ambiguous. Then if "tempb" occurs later and "b" is known to be replaceable by "buffer", the deabbreviation "temporary_buffer" will be the first guess. Analogizing can also infer deabbreviations more than three words long that the standard abbreviation rules cannot; for instance, if "tempg" is "temporary_global", "var" is "variable", and "nm" is "name", then "tempgvarnm" is "temporary_global_variable_name". With such analogizing, the system can become progressively more able to guess what the user means as it works on a program.

Acronyms are not recognized by the standard deabbreviation methods, so analogies are especially important for them, to recognize them as components of abbreviations. Acronyms make poor names since they are highly ambiguous, but they were rare in programs we studied.

Previous user replacements (whether generated by the program or by the user) can be very helpful in analogies. We give the user the option of keeping the replacements (both explicit and inferred) from earlier programs that were deabbreviated, so that modules of related programs can be treated together.

6. Experiments

To test the tool, 15 C programs were deabbreviated by the second author and 7 Prolog programs were deabbreviated by the first author. The programs were written by a variety of programmers for a variety of applications. CPU time averaged about 1.5 seconds per source-program word with semi-compiled Quintus Prolog running on a Sun SparcStation, with questions to the user, after initialization, coming at the comfortable rate of about one per every two seconds, although there was significant inter-program

variation. The C programs had a total of 20,990 symbols, all but 1096 distinct occurrences of which were a-priori acceptable (as either dictionary entries, numbers, or punctuation); of the 1096, 598 proposed deabbreviations were accepted by the user, 434 proposed deabbreviations were rejected, 274 were added as new words to the dictionary, 175 required explicit user replacements, and 49 short codes were left untouched. The Prolog programs had 5850 symbols, all but 605 distinct occurrences of which were a-priori acceptable; of the 605, 371 proposed deabbreviations were accepted by the user, 123 proposed deabbreviations were rejected, 9 were added as new words to the dictionary, 140 required explicit user replacements, and 85 short codes were left untouched. Thus the tool guesses acceptable replacements more than half the time, and appears to save users significant time in the task of replacing names in a program. These results also indicate that multiple possible deabbreviations are not common, and support our decision not to rank deabbreviations of a word.

We tested some tool output on 24 subjects, third-quarter students of a M.S. program in Computer Science. We used two programs written previously for other purposes, one in C of 373 symbols, and one in Prolog of 118. Half the subjects got the original C program and the deabbreviation of the Prolog program; the other half got the deabbreviation of the C program and the original Prolog program. We asked 10 multiple-choice comprehension questions in 20 minutes about the C program and 8 questions in 15 minutes about the Prolog program. Some questions asked about purposes ("Why are there four cases for `median_counts`?"), some about execution behavior ("What arguments to `median` should be bound before querying it?"), and some about exceptions ("What happens if the word being inserted into the lexicon is there already?"). Subject performance was better on both deabbreviated programs: 7.69/10 versus 7.17/10 for the C program, and 4.64/8 vs. 3.62/8 for the Prolog program. Using Fisher's 2-by-2 test, we confirmed that output of the deabbreviator gave higher comprehensibility at significance level 4.8%.

7. Conclusion

Our experience and experiments show that a deabbreviation tool with inferencing is easy to use and can

definitely improve the comprehensibility of a source program. It could be educational, since it requires the user to carefully describe program variables and procedures. But more importantly, it could help in the costly task of software maintenance.

8. References

C. P. Bourne and D. F. Ford, A study of methods for systematically abbreviating English words and names. *Journal of the ACM*, 8, (1961), 538-552.

R. Dale, Computer-based editorial aids. In *Recent developments and applications of natural language processing*, London: Kogan Page, 1989, 8-22.

K. Kukich, Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24, 2 (December 1992), 377-439.

K. Laitinen, Using natural naming in programming: feedback from practitioners. Proceedings of the Fifth Workshop of the Psychology of Programming Interest Group, Paris, December 1992. Also included in VTT Research Notes 1498, The principle of natural naming in software documentation, Technical Research Centre of Finland, 1993.

K. Laitinen and T. Mukari, DNN -- Disciplined Natural Naming. Proceedings of the 25th Hawaii International Conference on System Sciences, vol. II, 1992, 91-100.

J. L. Peterson, Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23, 12 (December 1980), 676-687.

J. M. Rosenberg, *McGraw-Hill dictionary of information technology and computer acronyms, initials, and abbreviations*. New York: McGraw-Hill, 1992.

B. Schneiderman, *Software psychology: human factors in computer and information systems*. Cambridge, MA: Winthrop, 1980.

L. M. Weissman, A methodology for studying the psychological complexity of computer programs.

Ph.D. thesis, Dept. of Computer Science, University of Toronto, 1974.

Distribution List

• Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

1

• Library, Code 52
Naval Postgraduate School
Monterey, CA 93943

1

Research Office
Code 08
Naval Postgraduate School
Monterey, CA 93943

1

Dr. Neil C. Rowe, Code CSRp
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943-5118

50

Mr. Russell Davis
HQ, USACDEC
Office of Naval Research
Attention: ATEC-1M
Fort Ord, CA 93941

1

Ralph Wachter, Code 333
Computer Science
Office of Naval Research
Ballston Tower One
800 North Quincy St.
Arlington, VA 22217-5660

1